[0015] **FIG. 1** describes the process of retrieving a property from a configuration file. At step **102**, an external program requests the value of a configuration property by passing in a configuration 'key' of the format 'config.section.name' (e.g. 'MyConfig.MySection.myVa1'). At step **104**, the configuration key is parsed, and the process checks to see if the configuration file identified by the 'config' part of the key has been loaded into memory (e.g. 'MyConfig.cfg').

[0016] At step **106**, the configuration file is loaded into memory. This consists of reading all properties in the configuration file and storing them in memory. At step **108**, the property value associated with the key is retrieved from memory. At step **110**, the property value is checked to see if it has a variable in it by looking for the string pattern '$x{y}', where x consists of one or more letters, and y consists of a string with zero or more recursively nested variables. At step **112**, the variable is extracted from the property string. If the 'y' portion of the variable contains recursively nested variables, then these are recursively resolved until the 'y' portion of the variable contains no variables. Then a variable-resolver (described in more detail below) associated with the string 'x' is used to resolve the value of the variable as plain text. This plain text value is then reinserted into the original property string, overwriting the previous variable. This process is repeated until all variables in the property are resolved to plain text.

[0017] At step **114**, the resulting plain-text property and returned to the external program. Optionally, the resolved property value can be stored in memory in place of the previous value in memory so that subsequent property retrievals do not have to re-resolve the variables.

[0018] The variable-resolver is a construct that accepts input text, and returns output text relevant to the input text passed in. In Java, this can be accomplished by implementing classes of the following interface:

```
public interface Var {
    public String resolve(String arg);
}
```

Each variable type defined in a configuration file must have an associated variables resolver (e.g. $C{y}CVar, $L{y}LVar, etc.).

[0019] The following variables have been defined that implement the dynamic nature in the configuration files.

[0020] $C{Config.Section.Name}—Implements the cross-referencing capabilities between configuration properties. The value of 'Config.Section.Name' is passed to the process defined in **FIG. 1** to resolve the value of the specified property.

$L{ResourceBundleProperty}—Implements internationalization support. The value of 'ResourceBundleProperty' refers to a property defined in a Java resource bundle.

[0021] $MB {expression}—Implements Boolean math support. The value of 'expression' is a Boolean mathematical expression, and the $MB variable resolves to the strings 'true' or 'false'. Example: $MB{(1>2||3<4) & false}'false'

[0022] $MI {expression}—Implements integer math support. The value of 'expression' is a mathematical expression, and the $MI variable solves that expression and returns the resulting integer as a string. Example: $MI{1+2*3}'7'

[0023] $MF {expression}—Implement float math support. The value of 'expression' is a mathematical expression, and the $MF variables solves that expression and returns the resulting float values as a string. Example: $MF{**3**/(1+1)}'1.5'

[0024] The flow of the process described in **FIG. 1**, known as the method "ConfigMgr.get( )", is as follows:

[0025] 1. User calls ConfigMgr.get("MyConfig1.MySection1.myVar1").

[0026] 2. ConfigMgr parses key.

[0027] 3. ConfigMgr looks for file MyConfig1.cfg in internal cache. If not found in the internal cache, then the file is loaded into the internal cache.

[0028] 4. ConfigMgr looks for MySection1 in MyConfig1.

[0029] 5. ConfigMgr looks for myVar1 in MySection1.

[0030] 6. ConfigMgr retrieves the value for myVar1 from memory and puts the value in VALUE.

[0031] 7. ConfigMgr calls VALUE=ConfigMgr.resolve(VALUE).

[0032] 8. ConfigMgr returns VALUE.

[0033] The flow of ConfigMgr.resolve(X) is as follows:

```
1.  Looks for existence of first $_{} variable. If none exist,
    returns X.
2.  Get contents of body of variable (i.e. $C{<body of variable>})
    and puts it in BODY.
3.  Calls BODY = ConfigMgr.resolve(BODY) to recursively resolve any
    internal variables.
4.  Depending on the variable...
    If ($C) {
        calls CVar.resolve(BODY), and replaces $C variable in X
        with the results.
    } else if ($L) {
        calls LVar.resolve(BODY), and replaces $L variable in X
        with the results.
    } else if ($MB) {
        calls MBVar.resolve(BODY), and replaces $MB variable in X
        with the results.
    } else if ($MI) {
        calls MIVar.resolve(BODY), and replaces $MI variable in X
        with the results.
    } else if ($MF) {
        calls MFVar.resolve(BODY), and replaces $MF variable in X
        with the results.
    }
5.  Go to step 1.
```

[0034] Passwords can also be stored, using the present invention, in dynamic configuration files thereby resulting in a reduced security risk. Using the present invention for password encoding functions as follows. If a "*" is added to the end of a property name in a configuration file, the value becomes encoded the first time the file is encountered through the ConfigMgr utility. Thus, if a file appears as follows: